Code generation for room acoustics simulations with complex boundary conditions using LIFT

Code Generation for Room Acoustics Simulations with Complex Boundary Conditions

Larisa Stoltzfus, Brian Hamilton, Michel Steuwer, Lu Li
University of Edinburgh, United Kingdom
{larisa.stoltzfus, brian.hamilton, michel.steuwer, lu.li}@ed.ac.uk

Christophe Dubach
McGill University, Canada
christophe.dubach@mcgill.ca

***Abstract*—The software and hardware landscape of high performance computing is expanding faster than computational scientists can take advantage of new frameworks and platforms. In an ideal world, simulation codes would be written once in a high-level manner and achieve high-performance anywhere, but the reality is more complicated. Currently, high-level solutions lack support for sophisticated physical models across different parallel backends. Existing solutions with appropriate support are low-level and, therefore, tied to a specific hardware target.**

We present an approach that tackles this problem with a modularized separation of concerns: a middle layer separates the management of generating low-level optimized code from a high-level programmable layer. In this paper, we describe how our contributions to this hardware-agnostic, middle-layer language provide functionality for complex room acoustics simulations, a type of Finite Difference Time Domain (FDTD) simulation using stencils which is representative of many other 3D wave models. We show that we are able to develop performance-portable codes for these types of models which leads to performance on par with tuned hand-written implementations. Furthermore, we show how this approach is used to develop both host and device side code for multi-kernel applications, as is required for room acoustics simulations with complex boundaries.

***Index Terms*—compilers, stencils, programming languages**

I. Introduction

High-level programming and code generation approaches promise high performance through the expression of applications in a convenient manner by developers. High-level program representation enables code generators to optimize for different hardware without having to manually rewrite or re-optimize applications. However, high-level approaches are often not chosen for application development and lower level approaches are preferred instead — despite their known drawbacks.

These high-level approaches often oversimplify the application domain by nicely abstracting out only the most intensive computational aspects. They focus on the most common simple use cases, which are easier to model and optimize for. However, applications have important corner cases which must be accounted for when developing abstractions, as these corner cases can be integral for achieving high performance.

In this paper, we investigate room acoustic simulations as an interesting representative of a broader class of FDTD simulations. Room acoustics simulations model the behavior of sound waves as they travel through an enclosed space and require parallelization to produce results in a timely manner. With applications in architectural acoustics and virtual reality [1], there is also a great interest running them at large scales on HPC systems [2]–[5]. The core computational pattern of these simulations is a stencil — a well known and widely studied pattern [6]. However, there are other important aspects to consider to express these simulations at a high-level and optimize performance across different platforms.

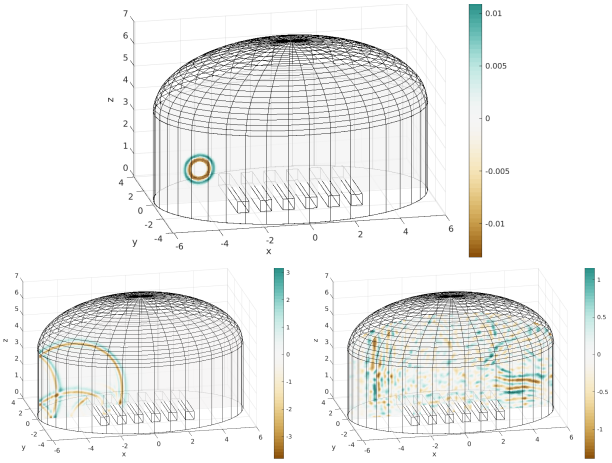

Fig. 1: Complex Boundary Conditions in Room Acoustics Simulations model reflections of sound waves from different surfaces in a non-cuboid room at three snapshots in time.

Complex boundary conditions modeling the reflections of sound waves from different surfaces, as shown in Figure 1, are an important component of realistic room acoustic simulations. To the best of our knowledge, modeling and optimizing these complex boundary conditions is not currently supported by existing high-level approaches such as Devito [7]. Supporting abstractions and code generation for complex boundary conditions is important, since over 20% of the simulation time can be spent computing the boundary alone as we will see in Section II.

This paper adds support for complex boundary conditions in the intermediate representation of the Lift data-parallel language and code generator. Lift has been shown to produce high performance code for simple stencils [8], [9]. By making only a few small additions to the code generator and its intermediate language, support for complex boundary conditions as found in 3D wave-based room acoustics simulations is provided. This highlights the advantages of using a pattern-based and extensible code generator such as Lift, which can be targeted by higher-level libraries or DSLs. Our paper also serves as a reminder of the importance of studying realistic applications instead of overly focusing on over-simplified benchmarks.

This paper makes the following contributions:

1) to the best of our knowledge, it models complex boundary conditions for 3D wave simulation in the intermediate representation of a high-level code generator;
2) it demonstrates that our performance-portable code generator achieves performance comparable to hand-written simulation codes across different GPUs.

II. Challenges of Acoustic Simulations with Complex Boundary Conditions

Acoustics simulations model sound waves propagating through a volume. This type of behavior can be simulated as a stencil computation where the volume is discretized into a grid of voxels. The value in each voxel represents the amplitude (or energy) of the wave propagating through space at a given time-step. In the simple case, at every simulation time-step the value of each voxel is updated as a function of its past neighbors' values at discrete times $t-1$ and $t-2$. The wave propagates outward in all directions until it hits the surface of an obstacle, *e.g.*, a wall or a bench as can be seen in Figure 1.

The goal of this paper is to show how high-level abstractions can be used to generate high-performance code for room acoustics simulations on GPUs. Before describing these high-level abstractions, this section reviews, step by step, how realistic acoustic simulation models are developed. It also takes a deeper a look at the intricate complexity of writing such an implementation, which will serve as a motivation for the rest of the paper.

A. Special Boundary Handling

Many stencil computations, *e.g.*, Gaussian blur, perform the same computation throughout the grid. However, acoustics simulations need to model a wave bouncing off obstacles, resulting in a different computation at the boundary. A key feature in this model is the *absorption* of some of the wave energy at the boundary. When the simulation processes a new time step, the reflection wave leaving the boundary will have less energy, which could be dependent on its frequency of oscillation. In order to prevent the wave from passing through the material, the points lying outside of the boundary are never updated.

Simple Example: We first consider a simple acoustics simulation which forms the basis of more advanced examples. The room boundary is composed of four walls, a ceiling and a floor. As is typical with stencil codes, the volume is zero-padded around the edge to prevent illegal memory accesses, forming a *halo*. The inputs are the simulation state at two time-steps, stored in arrays `prev` and `curr`. The output is the new simulation state is stored in array `next`. The size of each array is equal to the number of points in the volume plus the halo.

Listing 1 shows the implementation of this simple acoustic simulation in C. The computation of `nbr` on Lines 3–6 determines the number of neighboring points inside the boundary for a given point in the volume. Here `nbr` is calculated on the fly and used when computing the `next` values, as part of the physics simulated. Importantly, it is also used to determine if a points lies outside, inside or at the boundary. A stencil computation is calculated for the points lying inside or at the boundary

```
1  // for all x,y,z in the volume
2  int idx = z*Nx*Ny+(y*Nx+x);
3  int nbr = (x==1?0:1)+(y==1?0:1)+(z==1?0:1)
4       +(x==Nx-2?0:1)+(y==Ny-2?0:1)+(z==Nz-2?0:1);
5  if (x==0||y==0||z==0||x==Nx-1||y==Ny-1||z==Nz-1)
6    nbr = 0; // outside
7  if (nbr>0) { // inside or at boundary
8    double s = curr[idx-1]+curr[idx+1]
9              +curr[idx-Nx]+curr[idx+Nx]
10             +curr[idx-Nx*Ny]+curr[idx+Nx*Ny];
11   if ((nbr<6)) { // at boundary
12     double cf = 0.5*l*(6-nbr)*beta;
13     next[idx] = ((2.0-l2*nbr)*curr[idx]+l2*s
14                  +(cf-1.0)*prev[idx])/(1.0+cf);}
15   else // inside
16     next[idx] = ((2.0-l2*nbr)*curr[idx]
17                  +l2*s-prev[idx]);}
```

Listing 1: Simple Acoustic Stencil implementation in C [10]. A simple implicit boundary shape is used: a box.

in the current timestep on Line 8. Depending on whether the point is at the boundary or inside, the computation performed for the next voxel is different to account for the wall *absorption*.

B. Complicated Boundary Shapes

As shown above, a couple of Boolean formulas are sufficient to identify whether a point is inside, outside or at the boundary when considering a simple shape. However, in the case of the dome-shaped room from the introduction, it is not always possible to use such simple set of Boolean formulas. Instead, complicated boundary shapes require a dedicated data structure that encodes whether each point is inside, outside or at the boundary.

Updated Simple Example: For complicated shapes, the Lines 3–6 in Listing 1 is replaced by a lookup:

```
int nbr = nbrs[idx];
```

For each point in the volume, `nbrs` stores the number of neighbors lying within the boundary, which is pre-calculated for a given shape. A value of zero is used for points outside.

C. Boundary Handling Separation

Making acoustics simulations more realistic is linked to the physics at the boundary (where the wave bounces off). Simulating the process of propagating a wave through air is fairly straightforward and is usually not the focus of attention of acoustics experts. Therefore, the simulation is typically separated into two distinct phases: one for processing the volume (air) and one for processing the boundary (obstacles). This enables a modular software design, but also has performance benefits on GPUs as we will explain below.

The first simulation phase over the volume performs a stencil computation for points inside or at the boundary. This performs efficiently on GPUs as divergence is removed with most threads performing the same computation. The second simulation phase only handles points at the boundary. For each boundary point, values calculated by the first kernel are updated *in-place*, reusing parts of the stencil computation results from the prior phase. In the second phase the *absorption* discussed previously is also computed.

```
1  // kernel 1: volume handling
2  // for all x,y,z in the volume
3  int idx = z*Nx*Ny+(y*Nx+x); int nbr = nbrs[idx];
4  if (nbr>0) { // inside or at boundary
5     double s = curr[idx-1]+curr[idx+1]
6              +curr[idx-Nx]+curr[idx+Nx]
7              +curr[idx-Nx*Ny]+curr[idx+Nx*Ny];
8     next[idx] = ((2.0-l2*nbr[idx])*curr[idx]
9                  +l2*s-prev[idx]); }
10 ------------------------------------------------
11 // kernel 2: boundary handling (simple)
12 // for all i in [0; numBoundaryPoints[
13 int idx = boundaryIndices[i];
14 int nbr = nbrs[idx];
15 double cf = 0.5*l*(6-nbr)*beta;
16 next[idx] = (next[idx] + cf*prev[idx])/(1.0+cf);
```

Listing 2: Simple two-kernels approach supporting complex boundary shapes. The `nbrs` and `boundaryIndices` array contain information about each point.

```
1  // kernel 2: boundary handling (FI-MM)
2  // for all i in [0; numBoundaryPoints[
3  int idx   = boundaryIndices[i];
4  int nbr   = nbrs[idx];
5  int mi    = material[i];
6  double cf = 0.5*l*(6-nbr)*beta[mi];
7  next[idx] = (next[idx] + cf*prev[idx])/(1.0+cf);
```

Listing 3: Frequency-Independent (FI-MM) boundary handling in C [11] - `material` stores the material type for each boundary point, `beta` contains a special coefficient for each material.

Simple Two-Kernels Approach: This two-kernels approach is illustrated in Listing 2. As explained in the previous section, the points lying inside or at the boundary in the first kernel are identified by the pre-calculated `nbrs` array. Line 4 ensures that points outside of the boundary are not processed to prevent the propagation of the wave through obstacles.

The second kernel only processes the boundary and simulates the *absorption* process. It uses a second explicit data structure, `boundaryIndices`, which contains the indices of the points at the boundary. This two-kernel approach is efficient on GPUs; it is easily parallelizable and removes conditional branching which hinders performance. Note that the update of `next` on Line 16 reuses the computation from Line 8, removing the need to perform a full stencil computation in the second kernel.

D. Absorption with Multiple Materials

Modeling how different materials absorb wave energy achieves more realistic simulations. For instance, a cushion surface will absorb more sound energy than a concrete wall, resulting in a quieter, reflected sound. One such physical model is the *frequency-independent absorption* [11], where a material absorbs energy equally for all wave frequencies.

Frequency-Independent Absorbing Boundary (FI-MM): Listing 3 shows the second kernel for boundary handling with multi-material support (first kernel remains unchanged). The extra data structure `material` stores the material type at each boundary point. The computation of `cf` on Line 6 uses a `beta` coefficient specific to each material.

```
1  // kernel 2: boundary handling (FD-MM)
2  // for all i in [0; numBoundaryPoints[
3  double _g1[MB],_v2[MB]; // local temporaries
4  int idx = boundaryIndices[i];
5  int nbr = nbrs[idx];
6  int mi  = material[i];
7  double cf1 = l*(6-nbr);
8  double cf  = 0.5*cf1*beta[mi];
9  double _next = next[idx];
10 double _prev = prev[idx];
11 for (int b=0; b<MB; b++){ // for each ODE branch
12   ci = b*numBoundaryPoints + i;
13   _g1[m] = g1[ci];  _v2[m] = v2[ci];
14   _next -= cf1*BI[mi][b] *
15           (2.0*D[mi][b]*_v2[b]-F[mi][b]*_g1[b]);}
16 _next = ( _next + cf * _prev ) / ( 1.0 + cf );
17 next[idx] = _next;
18 // for each ODE branch
19 for (int b = 0; b < MB; b++) {
20   ci  = b*numBoundaryPoints + i;
21   double _v1 = BI[mi][b] *
22                (_next - _prev + DI[mi][b]*_v2[b]
23                 - 2.0*F[mi][b]*_g1[b]);
24   g1[ci] = _g1[b]+0.5*(_v1+_v2[b]);
25   v1[ci] = _v1;}
```

Listing 4: FD-MM boundary handling in C [11]

E. Boundary State

Real-life materials absorb certain frequencies more than others due to the presence of internal resonances. When these resonances are excited, the outgoing wave amplitude at the next time-step will be reduced to greater degrees. Modeling this behavior requires the use of a system of second-order ordinary differential equations (ODE) with multiple *ODE branches* [11], [12]. While we will not elaborate on the physics of this modeling, extra state information must be stored at the boundary. Intuitively, this state represents the internal vibration of the material structure over time.

Frequency-Dependent Boundary Handling (FD-MM): An implementation of this more accurate simulation is shown in Listing 4. As with FI-MM, this kernel only processes the points at the boundary with multiple materials support.

One of the main differences is the use of `g1` and `v2`, arrays storing values associated with the boundary. As seen on Lines 9–17, the `next` value is updated by processing and combining the information from these arrays.Values from `g1` and `v2` are reused later, so are first saved in temporary arrays, which could reside in registers or fast shared memory on a GPU. The state at the boundary is updated using the new value of `next`.

This final code is the most advanced — and realistic — acoustics simulation modeling we will evaluate in this paper. Our main goal is to show how such complex code can be expressed using high-level programming abstractions.

F. Complex Boundaries Handling Performance

As additional motivation, we briefly discuss the importance of efficient boundary handling. Figure 2 shows the percentage of a room acoustics simulation which is spent processing boundary elements (kernel 2) for a tuned implementation on a GPU. This boundary handling algorithm represents a computationally

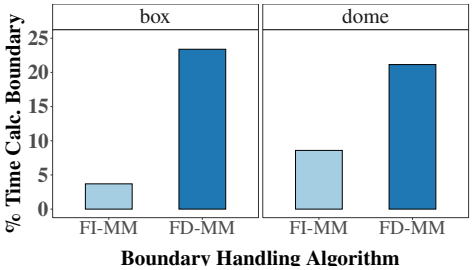

Fig. 2: Boundary handling % of total computation time for acoustic simulation of a dome and a box. Hand-written CUDA codes [11] running on an NVIDIA GTX780 GPU.

intense (but realistic) calculation involving several for-loops. Each loop performs memory reads and writes to several different arrays in global memory, accounting for a significant 20% of the time for this most realistic FD-MM implementation. As the complexity and realism of acoustics simulations will keep increasing, boundary handling can become a serious bottleneck, further motivating the process in this paper.

G. Summary

This section has reviewed how realistic acoustics simulations are modeled step by step. We have shown how these codes require extra data structures to retain states and describe boundaries. Additionally, we have outlined the need to perform in-place updates, handle multiple materials and store states for each boundary point. Current high-level existing approaches lack support to express such complex applications. For instance LIFT– the functional high-level code generator this work extends – lacks the following abilities:

- writing to memory locations selectively, (*i.e.*, in-place);
- producing multiple arrays of different sizes in one kernel;
- and generating code on the host side to automatically schedule multiple kernels.

The next section provides some technical background on the LIFT system and its limitations for expressing room acoustics simulations with complex boundary conditions, before addressing these in the subsequent section.

III. LIFT CODE GENERATOR OVERVIEW

This work extends the LIFT high-performance GPU code generator to implement room acoustics simulations with complex boundary conditions. LIFT [13] represents computations in an easy-to-extend functional pattern-based language. The LIFT internal representation (IR) is optimized by applying semantic-preserving rewrite rules encoding different optimization and implementation choices. Starting from a single program representation, different optimizations are applied for varying hardware targets. LIFT has previously demonstrated it can generate high-performance code across different application domains and hardware targets [9], [14], [15].

LIFT IR is not intended for directly writing applications, such as room acoustics codes. Instead, it is meant to be targeted by DSLs or libraries. This paper demonstrates it is possible to express complicated acoustics simulation in LIFT, opening the door for performance portable LIFT-based implementation of future and existing DSLs (or libraries).

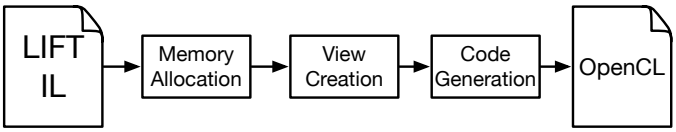

Fig. 3: LIFT Intermediary Language code generation workflow.

A. The LIFT Code Generator

The LIFT code generator performs a number of steps to lower LIFT's pattern-based IR to OpenCL code. The program is first rewritten, by lowering high-level patterns into low-level patterns, explicitly encoding how the pattern should be executed.

After rewriting, the code generation process starts which is detailed in Section III-A. First, the system determines where memory for temporary values must be allocated, if any. Then, compiler-intermediate data structures, called *views*, are created to capture memory access patterns. Finally, OpenCL code is generated. The following LIFT code example shows the summing of two arrays of float to produce an array of float:

```
1 fun(A: Array(Float, N), B: Array(Float, N) =>
2   mapSeq(p => p.get(0) + p.get(1)) o zip(A,B)
3 : Array(Float)
```

View Creation: Once memory allocation has been performed, the view creation stage builds for every expression in the LIFT program an input and output view. The view encodes information about the location of the data that is being manipulated. Here are three example views for the code given above:

```
1 inputView(p.get(0)) =
2   TupleAccessView(0, ArrayAccessView(i,
3     ZipView(MemView(A), MemView(B))))
4 inputView(p.get(1)) =
5   TupleAccessView(1, ArrayAccessView(i,
6     ZipView(MemView(A), MemView(B))))
7 outputView(p.get(0) + p.get(1)) =
8   ArrayAccessView((MemView(out),i)
```

For instance, the first `inputView` describes the location of the expression `p.get(0)`. This corresponds to an access to the first tuple component, of an access to the i^{th} element of the results of zipping two memory objects A and B.

Code Generation: The code generator then uses this information to determine the location of each read and write. The resulting C code for the example above would be:

```
1 fun(float * A, float * B, float_float_t * out) {
2   for (i=0; i<N; i++) {
3     float tmp1 = A[i];
4     float tmp2 = B[i];
5     float tmp3 = tmp1+tmp2;
6     out[i] = tmp3;    }                    }
```

where the C expression `A[i]` results from evaluating the `inputView` discussed above. The TupleAccessView will select the right memory `A` while the ArrayAccessView will select the right index `i` as can be seen on Line 3. A more detailed description of this process can be found in [15].

B. Stencil Computations in LIFT

Section II has shown that stencil computations are at the heart of room acoustic simulations. LIFT expresses simple stencils using a composition of three fundamental patterns, each modeling a different aspect of the stencil computation. An example is given below for a simple 1D stencil:

```
1 fun(A: Array(Float, N) =>
2   map(reduce(add, 0.0), slide(3, 1,
3     pad(1, 1, c, A))))
```

Slide creates the stencil neighborhoods while *map* applies the nested reduction to each neighborhood. *Pad* performs a naive boundary handling by enlarging the input on both sides according to a constant `c`. These alone are not enough to model the more complex boundary handling discussed in Section II.

IV. LIFT LANGUAGE AND COMPILER ADDITIONS

This section presents additions to LIFT necessary to handle the complex boundary handling introduced in Section II.

A. Host Code Multi-Kernel Management

As discussed in Section III, acoustics simulations are separated into two distinct phases: volume handling and boundary handling. This section explains how the LIFT compiler is extended to manage the generation of OpenCL host code. Given that both the OpenCL kernel code and host code are dialects of C, it is possible to reuse most of the existing LIFT code generator to create host code. Four primitives (shown in Table I) are added to the LIFT language to express the host code side of the application.

OclKernel: This first primitive wraps an OpenCL kernel into a LIFT expression. When the OpenCL host code generator encounters this primitive, it passes the inner expression to the OpenCL kernel code generator. Then it emits OpenCL host code to set the kernel argument and trigger its execution. The kernel argument is set using the view system explained previously.

ToGPU/ToHost: This pair of primitives manages the data transfer between the host and the GPU. From a semantic point of view, they behave like an identity function. However, when encountering these primitives, the host code generator produces an OpenCL call to move data to the GPU or host.

WriteTo: This primitive dictates the location of where the result of an expression should be written to. As we will see later, this primitive is useful when performing in-place updates. Its implementation is discussed in the next section.

B. GPU Device In-Place Updates

We now turn our attention to the OpenCL GPU kernel code generator. The listing below shows a simplified in-place boundary handling illustrating the core issue we are addressing:

```
1 for(i=0; i<boundaryPointIndices.length; i++) {
2   int idx = boundaryPointIndices[i];  // (A)
3   float newVal = gridPoints[idx];     // (B)
4   gridPoints[idx] = f(newVal);        // (C) }
```

This code requires the following algorithmic functionality:

A) Read each index value from an array;
B) Read a new value at the index location from an array;
C) Update *in-place* an existing array at the specific index with `f` of the new value.

Generating in-place updates in LIFT requires the new primitives *Concat*, *Skip* and *WriteTo*. *WriteTo* is used to specify to write updates at the same input array, preventing the allocation of an output buffer that would happen automatically in the memory allocator. The code generator behaves as if it is writing to an entire array at each iteration, but behind the scenes it only writes values at `idx`. This is accomplished by *concatenating* an expression which *skips* `idx` elements, essentially producing an array of `idx` offsets together with the new data to write.

1) New GPU Kernel LIFT *Primitives:*

WriteTo: The behavior of this primitive is the same as the host primitive introduced previously. During view construction, it sets the outputView of the second argument to the inputView of the first argument. As in the example in Table I, we assume the inputView of the first argument, `in`, is in memory (`ViewMem(in)`). *WriteTo* sets the outputView of the second argument to `ViewMem(in)`. In the *Map* function, the outputView of `add2` becomes `ViewArrayAccess(i,ViewMem(in))`. This results in assignment to `a[i]`, instead of to a newly allocated output buffer.

Concat: As shown in Table I, *Concat* takes in one or more arrays and returns the concatenation of those arrays as a single array. From the code generator point of view, during view construction, a new view called *ViewOffset* is created for each of the arguments in the *Concat*. For a given argument, the offset is set to the sum all the previous argument lengths and is added to the index when the array is accessed. For instance, the output view for `mul3` in the example in Table I is `ViewAccess (i1,ViewOffset(N0,ViewMem(out)))`, where `i1` is the `Map` iteration variable and `N0` the length of A.

Concat is a commonly used function can be utilized for a variety of use cases. Most importantly, in terms of boundary handling, we are now able to write in-place. However, *concat* can also re-pad arrays for iteration after a stencil is calculated, which is useful in tandem with the host code generator.

ArrayCons: The `ArrayCons` primitive enables the creation of arrays out of a single element that is repeated `n` times. As we will see shortly, this primitive is useful when used in conjunction with *Concat* to perform in-place updates.

Skip: This last primitive is, perhaps surprisingly, a *no-op*. The primitive is parameterized with a type `T` and, as can be see in Table I, it returns an array of `T` of length `i`. However, its semantic is such, that the code generator produces no code. Instead, this primitives influence the view construction mechanism so as to introduce an offset when writing to the output array. The next section will make it more clear why this primitive is useful.

2) Combining Primitives for In-Place Updates: We are now able to show how all these primitives are used to enable in-place updates in LIFT. Consider the following listing where we are mapping over an array of indices:

Primitives	Argument types	Ret. type	Lift example	Generated code	Platform
OclKernel	$(f : ([T_1]_{N_1}, [T_2]_{N_2}, ...) \rightarrow [U]_M)$, $in_1 : [T]_{N_1}, in_2 : [T_2]_{N_2}, ...)$	$[U]_M$	OclKernel(kernel,in)	kernel.setArg(0,in); enqueueNDRangeKernel(kernel,...);	host
ToGPU	$(in : [T]_N)$	$[T]_N$	ToGPU(in)	enqueueWriteBuffer(..., in);	host
ToHost	$(in : [T]_N)$	$[T]_N$	ToHost(in)	enqueueReadBuffer(in, ...);	host
WriteTo	$(to : [T]_N, in : [T]_N)$	$[T]_N$	WriteTo(in, Map(add2, in))	for(i=0;i¡N;++i) in[i] = add2(in[i]);	host & device

Concat	(arr_1: $[T]_{N_1}$, arr_2: $[T]_{N_2}$, ...)	$[T]_{\sum_{N_i}}$	Concat(
			Map(add2,A),
			Map(mul3,B))
				for(i0=0¡i0¡N1;i0++)
				out[i0]=add2(A[i0]);
				for(i1=0¡i1¡N2;i1++)
				out[i1+N1]=add2(B[i1]);
					device

ArrayCons	(e: T, n: int)	$[T]_n$	Map(id,ArrayCons(6,3))	for (int i=0;i¡3;i++) out[i] = 6;	device

Skip	$<T>(i$: $int)$	$[T]_i$	Concat(
			Skip$<int>$(n),
			Array(1,2,3))
				out[n] = 1;
				out[n+1] = 2;
				out[n+2] = 3;
					device

TABLE I: New Lift primitives

```
1 Map(idx => {WriteTo(input,
2     Concat(Skip(float,idx),
3            f(ArrayCons(input[idx],1)),
4            Skip(length(indices)-1-idx) ) )
5 }) << indices
```

As can be seen, three arrays are concatenated in this listing. The first one, of length `idx` is a dummy array whose sole purpose is to ensure that the outputView of the next element written by the concatenation is offset by `idx`. Then comes the in-place update, which will be performed at the offset of `[idx]`. Finally, a third dummy array is introduced to ensure that the return array from *concat* appears to be of the same length as the original array updated in place. From a type point of view, this LIFT expression looks like it is producing an array of rows.

However, if we look at the code generated (shown below), at every iteration of the loop from the *map* a single element is written out to the exact same `input` array.

```
1 for (int idx=0; i<indices.length; i++) {
2   // no-op (Skip(float,idx)
3   input[idx] = f(input[idx])
4   // no-op (Skip(length(indices)-1-idx)) }
```

The next section builds on this simple example to show how room acoustics boundary handling can be expressed in LIFT.

V. EXPRESSING ROOM ACOUSTIC BOUNDARY HANDLING IN LIFT

This section shows the room acoustics codes with complex boundary handling from Section II expressed in the extended LIFT language and code generator.

A. Expressing the Host Application

Using host code primitives, we can express the orchestration of OpenCL kernel launches and data movements functionally. Listing 5 shows how the host primitives introduced in Section IV combine to express multi-kernel acoustic simulations.

The host application has several input arguments, initially all stored in host memory. *ToGPU* is used to transfer the inputs to GPU memory. The application first launches the kernel performing the update of the grid volume ignoring the boundary handling (`volume_handling_kernel`) on line 4 and passes

```
1  (boundaries,neighbors,MIb,beta,
2   prev1_h,prev2_h) =>
3    val prev2_g = ToGPU(prev2_h)
4    val next_g =  OclKernel(
5                    volume_handling_kernel,
6                    ToGPU(prev1_h), prev2_g)
7    ToHost(
8      WriteTo(next_g,
9        OclKernel(boundary_handling_kernel,
10         ToGPU(boundaries),
11         ToGPU(neighbors), ToGPU(MIb),
12         ToGPU(beta),
13         next_g, prev2_g) ))
```

Listing 5: The executing of two kernels of an acoustic simulation updating the volume and the boundaries is orchestrated by host primitives in LIFT

the processed volume to the second kernel performing the complex boundary handling (line 10). Because the second kernel uses the output of the first kernel, a synchronization is generated after the first kernel is invoked. The use of the *WriteTo* primitive on line 9 indicates the boundary handling kernel will update the volume it is operating on in-place. Finally, the output of the boundary handling is transferred back to host memory. For an actual application the two kernels are executed iteratively.

B. Room Acoustic Simulation with Naive Frequency-Independent Boundary Handling (FI)

Prior work [9] has shown how to express room acoustic simulations in LIFT with a naive boundary handling strategy. This does not model the physical properties of boundaries correctly and, therefore, is of less interest to scientists. The LIFT code for this naive room acoustic simulation is shown in Listing 6. Here the stencil and boundary handling computation are handled in the same kernel. To create the stencil part of the code, the primitives *slide* and *pad* are used to create neighborhoods. Then a *map* is used to iterate over these neighborhoods in parallel. Inside this iteration, a naive form of boundary handling is performed by simply replacing missing values at the boundary with a constant using *pad*.

For the complex boundary handling discussed in the next two sections, a separate kernel performs the computations

```
1  acousticStencil(grid_{t-1},grid_t) {
2   map_3(m -> {
3     val sumGrid_{t-1} = m.1[0][1][1]
4     + m.1[1][0][1] + m.1[1][1][0]
5     + m.1[1][1][2] + m.1[1][2][1] + m.1[2][1][1]
6     val numNeighbor = m.2
7     return getCoeff(m.2, CSTloss1, 1.0f) *
8       ((2.0f-CST_{l2} * numNeighbor)*m.1[1][1][1]+
9        CST_{l2} * sumGrid_{t-1} -
10       getCoeff(m.2, CSTloss2, 1.0f) * m.0) },
11  zip_3(grid_t,slide_3(3, 1, pad_3(1,0,grid_{t-1})), array
       _3(m,n,o,computeNumNeighbors))) }
```

Listing 6: Acoustic simulation expressed in LIFT

```
1  ( boundaryIndices, nbrs, material,
2    beta, next, prev ) => {
3  (Map(fun(tup => {
4    val idx = Get(tup,0) // index
5    val nbr = Get(tup,1) // neighbors
6    val m = Get(tup,2) // material index
7
8    val betaVal = ArrayAccess( m ) << beta
9    val nextVal = ArrayAccess( idx ) << next
10   val prevVal = ArrayAccess( idx ) << prev
11   val lh = 0.5f * l
12   val cf = mult(multIF(nbr,betaVal),lh)
13   val boundaryUpdate =
14      boundaryHandle(nbr, nextVal, prevVal, cf)
15   Concat(
16     Skip( Float, idx),
17     Map(id) << ArrayCons(boundaryUpdate,1),
18     Skip( Float, N-1-idx ))
19 })) << Zip(boundaryIndices, nbrs, material)) }
```

Listing 7: Frequency-Independent (FI-MM) Boundary Handling in LIFT

at the boundary as described in Section V-A. The update of the volume remains similar to Listing 6, using the primitives introduced for stencil calculations — *map*, *pad*, *slide* — which are described more in Section III-B.

C. Frequency-Independent Boundary Handling (FI-MM)

Listing 7 presents the LIFT kernel expression for the FI-MM boundary handling, where boundaries are only updated at select points whose indices are found in the array *boundaryIndices*. Each index in the array is stored in a private variable called *idx*. The boundary calculation is performed on Line 14, with values gathered from the input on Lines 4–12. The original grid is then updated using *concat* on Lines 15–18. This update happens in-place due to the orchestrating host code that sets the output to be the same as the `next` input.

In the *concat* are two *skips* and an array value wrapped in an `ArrayCons` which are written on Line 17. The first *skip* on Line 16 produces the offset required to write `boundaryUpdate` to the correct memory location. The second *skip* on Line 18 retains the correct size of the output in LIFT's *views*. These three expressions combined enable LIFT to perform an in-place update at the correct memory location.

```
1  ( boundaryIndices, nbrs, material,
2    next, prev, vel_next, vel_prev, g1 ...) => {
3   Map(fun(tup => {
4    val uValUpdated = ...
5    val vel_nextValUpdated = ...
6    Tuple(
7      WriteTo(ArrayAccess(next,idx)) <<
8        divide(boundaryConstant, uValUpdated),
9      WriteTo(g1) o Map( fun( tup2 =>
10       calculateG1Update(
             Get(tup2,0), Get(tup2,1), Get(tup2,2),
11         cst_Ts)))  << Zip( vel_nextValUpdated
               , g1MbArray,vel_prevArr),
12     WriteTo(vel_next) << vel_nextValUpdated )
13  })) << Zip(boundaries, neighbors,
14            vel_next, vel_prev, g1) }
```

Listing 8: Frequency-Dependent (FD-MM) Boundary Handling in LIFT

D. Complex Frequency-Dependent Boundary Handling (FD-MM)

The most complex boundary conditions discussed in Section II maintain states at each boundary point to model resonating physical materials. Listing 8 shows the LIFT expression for this boundary handling. As this algorithm is much more involved than the previous two, we only show the overall structure and output that is written and leave out some details for clarity.

From a code generation perspective, the main difference in algorithmic complexity between this boundary handling algorithm and FI-MM — beyond the extra memory accesses and computations performed — is that three input arrays are written to in-place. These writes must be wrapped in a *tuple* upon return in order for LIFT to write to them correctly, as seen on Line 8. This allows for memory writes to be re-routed behind the scenes to the right output array using the LIFT view system. The presence of the `WriteTo` ensures that the input arrays `next`, `g1` and `vel_next` are updated in place.

In this section we have seen how the primitives introduced in Section IV are used to model complex boundary conditions in the LIFT high-level code generator. The host code primitives orchestrate the execution of multiple kernels. The additions to the device code allow for more fine-grained control on output locations, as well as multiple output functionality in a single functional program. Next, we explore the performance achieved when generating parallel GPU code from these representations.

VI. EXPERIMENTAL SETUP

Platforms and Measurements: Experiments are conducted using single and double precision floating points on: a GeForce GTX 780 with OpenCL 1.2 (361.42) and CUDA 8.0.20; an AMD Radeon HD 7970 with OpenCL 1.2 AMD-APP (1912.5); a GeForce GTX TITAN Black with OpenCL 1.2 (375.66) and CUDA 8.0.0; and an AMD Radeon R9 295X2 with OpenCL 1.2 AMD-APP (1598.5). Although OpenCL also runs on other platforms (ie. CPUs), this paper focuses on GPU results as they can provide much higher performance given the large computational requirement to simulate realistic rooms. The medians of

X Dim	Y Dim	Z Dim	B. Pts Dome	B. Pts Box
602	402	302	690,624	1,085,208
336	336	336	376,808	673,352
302	202	152	172,256	272,608

TABLE II: Room Sizes

Platform	Memory GB/s	Performance SP GFLOPS
NVIDIA GTX 780	288	3977
AMD Radeon HD 7970	288	4096
NVIDIA TITAN Black	337	5120
AMD Radeon R9 295X2	320	5733

TABLE III: Platforms and Hardware Metrics used

2000 executions are reported using the OpenCL profiling API with a standard deviation of 0.003 milliseconds. Only running times of each kernel are reported. More information about the hardware used can be found in Table III.

Benchmarks and Baseline: The stencil and boundary handling benchmark FI used in Section VII-A comes from the work done by Webb [10] and the boundary handling benchmarks FI-MM and FD-MM used in Section VII-B come from the work done by Hamilton et. al [11], both of which were originally written in CUDA. In order to compare across non-NVIDIA platforms a comparable, handwritten OpenCL version is used in this evaluation. This OpenCL baseline has been tuned and verified to produce on par or better performance than the CUDA version on NVIDIA platforms. All benchmarks have been hand-tuned by workgroup size and the best result is reported. Three room sizes have been evaluated on both a dome and a box shaped room and the dimensions and number of boundary points in each shape are described in Table II.

Throughput Metric: All results are presented in terms of million of update per second. This allows for a fair comparison across platforms and sizes without obfuscating timings.

VII. Evaluation

In this section we explore results comparing LIFT-generated code to handwritten versions of room acoustics codes. We present results for three different types of room acoustics simulations as introduced in Section II: 1) Frequency-independent (constant boundary handling), 2) Frequency-independent multiple material (FI-MM), 3) Frequency-dependent (FD-MM). Although all boundary-handling is normally run with complementary stencil computations, only the first result (frequency-independent) includes the stencil calculation in the reported results. For the FI-MM and FD-MM algorithms, we focus on the boundary handling that is performed in separate kernels.

A. Recap of Performance Results Naïve Frequency-Independent (FI) Boundary Handling Kernels

Previous studies [8], [9] have shown that LIFT produces comparable results for basic frequency-independent boundary handling room acoustics simulations. Figure 4 shows LIFT performing comparably with a hand-optimized reference

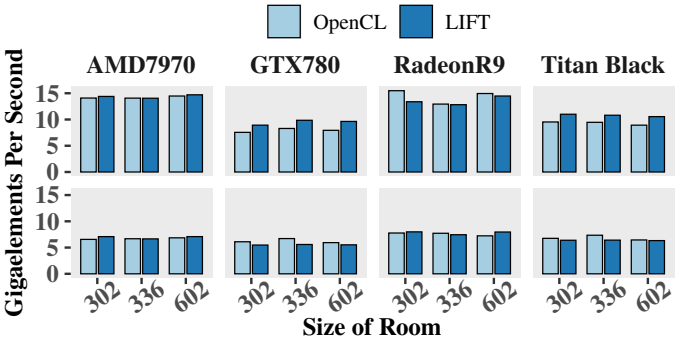

Fig. 4: Throughput of LIFT-generated codes versus manually written GPU code for room simulations with FI boundary handling in single (top) and double (bottom) precision. Raw runtime values are shown in Table IV.

version across four different GPUs for the same sized boxes used in the rest of the evaluation. Only box shaped rooms are shown because this benchmark can only handle cuboid shapes.

B. Performance of Complex Boundary Handling Kernels

This section presents performance comparisons for the two more involved boundary handling kernels that LIFT can automatically generate. As described in Section II, the FI-MM boundary handling algorithm accounts for multiple materials at the walls using more arithmetic operations at the boundary. FD-MM is the more complicated boundary handling kernel retaining multiple states at the boundaries.

1) Frequency-Independent Absorbing (FI-MM) Boundary Handling: This kernel takes in the indices of boundary values and iterates only over these and updates values in-place. A few computations and memory accesses are required in this algorithm in order to determine which material a given point is in and how many neighbors it has. Figure 5 shows that LIFT achieves performance on par with the manually written and tuned version.

One would expect to see similar performance for all three room sizes given the normalized throughput metric. However, the *336*-sized room achieves a smaller throughput in part because it is uniform in all dimensions, whereas the other two dimensions are cuboids with the largest dimension along the x-axis. Therefore, fewer continuous memory accesses are available along uniform-shaped boundaries so comparative performance is slower. This also explains why the box shape achieves overall better performance than the dome. A substantial difference also shows up between the LIFT and handwritten versions for double values on the two NVIDIA platforms. This discrepancy can be accounted for by the original benchmark using a hard-coded array of values in private memory, which is instead passed in as a parameter in the LIFT version.

2) Frequency-Dependent (FD-MM) Boundary Handling: Lastly, we show results for the FD-MM boundary handling kernel, which has frequency-dependent boundary conditions. FD-MM is more complex than FI-MM in that it uses ODE-branches, which translates into extra inputs, computations and

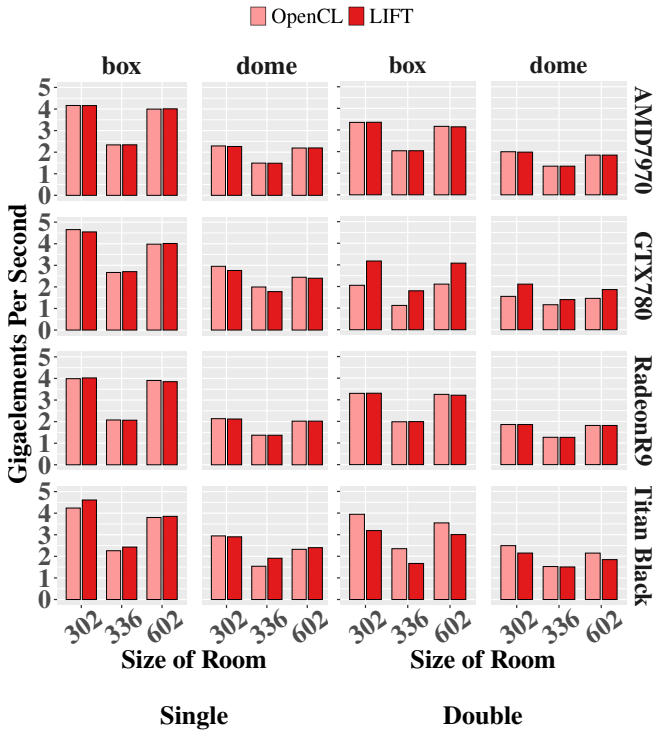

Fig. 5: Throughput of LIFT-generated to handwritten codes for FI-MM box and dome shapes in single (left) and double (right) precision. Raw runtime values are shown in Table V.

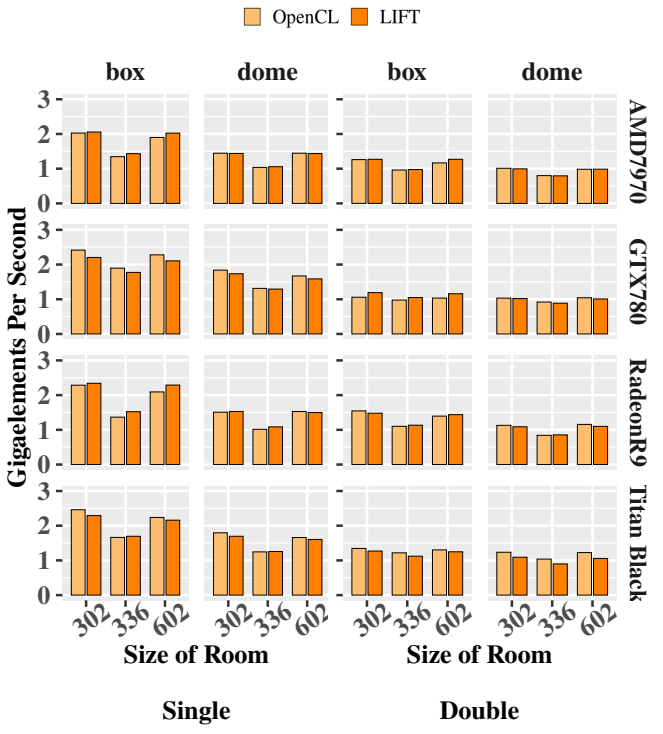

Fig. 6: Throughput of LIFT-generated to handwritten codes for FD-MM (branch value of 3) for box and dome shapes in single (left) and double (right) precision. Raw runtime values are shown in Table VI.

memory writes. This results in a more accurate simulation, but leads to a much lower throughput overall.

Figure 6 shows the relative performance of the LIFT-generated kernels versus the original benchmark for the FD-MM boundary-handling algorithm for a box and a dome. Comparable results are achieved with the hand-written version on both NVIDIA and AMD platforms. Just as for FI-MM, we see a dip in throughput for the *336*-sized room for similar reasons.

A notable difference between the FI-MM results in Figure 5 and the FD-MM results in Figure 6 is that the FD-MM shows a much bigger difference between single and double precision. This is because of the differences in memory accesses and computations in this algorithm. This FD-MM algorithm performs 45 memory accesses and 98 floating-point operations per update. The previous FI-MM version performs 6 memory accesses for only 7 computations per update.

VIII. Beyond Room Acoustics Simulations

There are other physical simulations with comparable properties that would benefit from the extensions to LIFT presented in this paper. In particular, other 3D wave models derived from FDTD numerical methods including in particular geophysical models like reverse-time migration [16] and ground penetrating radar [17] are programmed similarly. Reverse-time migration is a seismic imaging method used to model complicated subsurface forms using the wave equation. Ground penetrating radar models electromagnetic waves through different types of surfaces, which is applicable in fields ranging from structural engineering to medicine. Both of these other models use stencils and have PMLs (Perfectly Matched Layers) as boundary conditions, which also handle multiple materials and thus can be as complex as the boundary conditions seen in this paper.

As they have similar stencil calculations and complex boundary functionality, these models could readily reuse functionality introduced in this paper and be implemented in LIFT. While the only room acoustics code that requires multiple array updates in place is the FD-MM boundary handling algorithm, geophysical modelling such as ground penetrating radar and reverse-time migration require updating multiple arrays for the main volume calculation. This is because electromagnetic waves simulation requires modelling electric and magnetic fields separately, as well as updating each dimension independently, leading to six separate arrays being updated. These are all updated in-place as they are iterated over in a similar manner as room acoustics models due to the nature of finite difference simulations. As such, functionality for writing to arrays in-place is even more critical to these codes, as volume calculations still make up the vast majority of these algorithms total computation time.

IX. Related Work

Generic Code Generation Frameworks Many code generators have similar functionality to LIFT, including Delite [18], Accelerate [19], StreamIt [20] and Spiral [21], which aim to simplify GPU programming through higher level abstractions. However, many of these are less flexible in terms

of optimizations, do not fully provide performance portability or currently only support limited domains, such as Spiral for DSPs. Delite [18] is the most similar framework to LIFT and also uses a suite of parallel patterns which are compiled and optimized by a single backend into high-performance code. However LIFT is more extensible - allowing for optimizations to be explored and providing an IR for DSLs to compile into, as well as being extensible to new backends.

Stencil-Focused DSLs and Libraries Stencils are a widely targeted type of algorithm for DSLs (Domain Specific Languages) and skeleton frameworks and libraries. Some stencil-specific DSLs include Snowflake [22], StencilGen [23] and others [24], [25]. Skeleton frameworks and libraries supporting stencils include SkePU [26], SkelCL [27], MUESLI [28], and PASTHA [29]. Many of these solutions fall short however when it comes to complicated models as focusing on optimizing the stencil portion is more lucrative.

The types of stencils these frameworks focus on are too simple to accommodate complex shapes and boundaries found in real-world physical simulations and additionally produce low-level, optimized code much earlier in the compilation process than LIFT. Additionally these frameworks and languages are often limited to a particular domain or rely on heuristics or hard-coded or stencil-specific implementations. LIFT is specifically designed as an intermediate representation between high-level abstractions and low-level optimizations, capable of handling domains beyond stencils. None of these other solutions create a separation of concerns in the same manner.

Stencil-Focused Compilers and Code Generators There are also many compilers and code generators that focus on stencil algorithms, including Polly [30], Pochoir [31], PATUS [32], Pencil [33], PolyMage [34] and Halide [35]. Polly [30] implements the polyhedral model in LLVM IR to detect optimal loop transformations and Pencil [33] is an IR framework which also implements the polyhedral model and is intended to be targetted by DSLs. Pochoir [31] is a stencil compiler designed to target multi-core machines and PATUS [32] is an autotuner and compiler which uses domain and hardware specific heuristics to optimise stencil codes for CPUs and NVIDIA GPUs. PolyMage [34] and Halide [35] are two frameworks focusing on generating optimised codes for image processing stencils, but are limited in functionality beyond these specific stencils. These frameworks all focus on optimizing the main stencil computation and ignore complicated boundary conditions like those found in room acoustics simulations.

Frameworks Targeting 3D Wave Models Frameworks do exist which specifically target physical simulations with PDEs including Firedrake [36], Exastencils [37], Saiph [38], Devito [7] as well as many others [39]–[42]. Firedrake, Exastencils and Devito focus on abstractions at the mathematical level. Saiph and Devito can handle complicated boundary conditions, however both only target specific backends and neither currently supports the frequency-independent wave modeling which are described in this paper. LIFT specifically reuses functionality and can target multiple backends, giving it much greater flexibility beyond room acoustics simulations.

X. CONCLUSIONS

This paper has shown how a simulation with a sophisticated physical properties is expressible with a high-level data-parallel language. The room acoustic simulations studied use complex boundary conditions to model the physical properties of materials absorbing some of the energy of the sound wave. Existing high-level code generators lack support for these applications. We have extended the pattern-based code-generator LIFT with a few additions enabling the generation of high-performance GPU code across multiple GPU architectures achieving performance on-par with manually tuned code. The extended LIFT code-generator could be targeted by DSLs simplifying writing scientific applications to bring the performance benefits to end-users.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their helpful feedback. This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council [grant EP/L01503X/1] and the University of Edinburgh. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

REFERENCES

[1] M. Vorländer, *Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality*. Springer Science & Business Media, 2007.

[2] N. Morales, R. Mehra, and D. Manocha, "A parallel time-domain wave simulator based on rectangular decomposition for distributed memory architectures," *Applied Acoustics*, vol. 97, pp. 104–114, 2015.

[3] J. Saarelma, J. Califa, and R. Mehra, "Challenges of distributed real-time finite-difference time-domain room acoustic simulation for auralization," in *AES International Conference on Spatial Reproduction-Aesthetics and Science*. Audio Engineering Society, 2018.

[4] N. Raghuvanshi and J. Snyder, "Parametric directional coding for precomputed sound propagation," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–14, 2018.

[5] A. Melander, E. Strøm, F. Pind, A. Engsig-Karup, C.-H. Jeong, T. Warburton, N. Chalmers, and J. S. Hesthaven, "Massive Parallel Nodal Discontinuous Galerkin Finite Element Method Simulator for Room Acoustics," Tech. Rep., 2020.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The Landscape of Parallel Computing Research: A View From Berkeley," 2006.

[7] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P. H. Kelly, G. J. Gorman, and F. J. Herrmann, "Architecture and performance of devito, a system for automated stencil computation," *arXiv preprint arXiv:1807.03032*, 2018.

[8] L. Stoltzfus, B. Hagedorn, M. Steuwer, S. Gorlatch, and C. Dubach, "Tiling optimizations for stencil computations using rewrite rules in Lift," *TACO*, vol. 16, no. 4, pp. 52:1–52:25, 2020.

[9] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with Lift," in *CGO*. ACM, 2018, pp. 100–112.

[10] C. Webb, "Parallel Computation Techniques For Virtual Acoustics And Physical Modelling Synthesis," Ph.D. dissertation, University of Edinburgh, 2014.

[11] B. Hamilton, C. Webb, N. Fletcher, and S. Bilbao, "Finite difference room acoustics simulation with general impedance boundaries and viscothermal losses in air: Parallel implementation on multiple gpus," in *Proc. Int. Symp. Musical Room Acoust*, 2016.

[12] S. Bilbao, B. Hamilton, J. Botts, and L. Savioja, "Finite volume time domain room acoustics simulation under general impedance boundary conditions," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 1, pp. 161–173, 2016.

[13] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code," *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 205–217, 2015.

[14] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach, "Performance portable GPU code generation for matrix multiplication," in *GPGPU@PPoPP*. ACM, 2016, pp. 22–31.

[15] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: a functional data-parallel IR for high-performance GPU code generation," in *CGO*. ACM, 2017, pp. 74–85.

[16] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 79–84.

[17] C. Warren, A. Giannopoulos, and I. Giannakis, "gprmax: Open source software to simulate electromagnetic wave propagation for ground penetrating radar," *Computer Physics Communications*, vol. 209, pp. 163–170, 2016.

[18] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A Compiler Architecture For Performance-Oriented Embedded Domain-Specific Languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 134, 2014.

[19] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising Purely Functional GPU Programs," in *ICFP 2013*. New York, NY, USA: ACM, 2013, pp. 49–60.

[20] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software Pipelined Execution Of Stream Programs On GPUs," in *CGO*. IEEE, 2009, pp. 200–209.

[21] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[22] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox, "Snowflake: A lightweight portable stencil dsl," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 795–804.

[23] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[24] R. Membarth, F. Hannig, J. Teich, and H. Köstler, "Towards Domain-Specific Computing For Stencil Codes In HPC," in *SCC 2012*. IEEE, 2012, pp. 1133–1138.

[25] S. Kamil, D. Coetzee, S. Beamer, H. Cook, E. Gonina, J. Harper, J. Morlan, and A. Fox, "Portable Parallel Performance From Sequential, Productive, Embedded Domain-Specific Languages," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 303–304.

[26] J. Enmyren and C. Kessler, "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems." Baltimore, Maryland: HLPP '10: Proceedings of the fourth international workshop on High-level parallel programming and applications, Sep. 2010.

[32] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 676–687.

[27] M. Steuwer, M. Haidl, S. Breuer, and S. Gorlatch, "High-Level Programming of Stencil Computations on Multi-GPU Systems Using the SkelCL Library," *Parallel Processing Letters*, vol. 24, no. 3, Sep. 2014.

[28] H. Kuchen, "A Skeleton Library," in *Euro-Par 2002*. Springer, 2002, pp. 620–629.

[29] M. Lesniak, "PASTHA: Parallelizing Stencil Calculations In Haskell," in *Proceedings Of The 5th ACM SIGPLAN Workshop On Declarative Aspects Of Multicore Programming*. ACM, 2010, pp. 5–14.

[30] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[31] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir Stencil Compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 117–128.

[33] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema *et al.*, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 138–149.

[34] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization For Image Processing Pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 429–443.

[35] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language And Compiler For Optimizing Parallelism, Locality, And Recomputation In Image Processing Pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[36] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *ACM Trans. Math. Softw.*, vol. 43, no. 3, pp. 24:1–24:27, 2016.

[37] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter *et al.*, "Exastencils: Advanced stencil-code engineering," in *European Conference on Parallel Processing*. Springer, 2014, pp. 553–564.

[38] S. Macià, S. Mateo, P. J. Martínez-Ferrer, V. Beltran, D. Mira, and E. Ayguadé, "Saiph: Towards a dsl for high-performance computational fluid dynamics," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2018, p. 6.

[39] P. Bastian, M. Blatt, C. Engwer, A. Dedner, R. Klöfkorn, S. Kuttanikkad, M. Ohlberger, and O. Sander, "The Distributed And Unified Numerics Environment (DUNE)," in *Proc. Of The 19th Symposium On Simulation Technique In Hannover*, 2006.

[40] T. Brandvik and G. Pullan, "SBLOCK: A Framework For Efficient Stencil-Based PDE Solvers On Multi-Core Platforms," in *Computer And Information Technology (CIT), 2010 IEEE 10th International Conference On*. IEEE, 2010, pp. 1181–1188.

[41] E. Schnetter, M. Blazewicz, S. R. Brandt, D. M. Koppelman, and F. Löffler, "Chemora: A pde-solving framework for modern high-performance computing architectures," *Computing in Science Engineering*, vol. 17, no. 2, pp. 53–64, 2015.

[42] C. Osuna, T. Wicky, F. Thuering, T. Hoefler, and O. Fuhrer, "Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications," *Supercomputing Frontiers and Innovations*, vol. 7, no. 2, pp. 79–97, 2020.

APPENDIX

TABLE IV: Median run time values for naïve frequency-independent data in Figure 4. Times are reported in milliseconds.

Platform	Version	Size	Single (ms)	Double (ms)	Platform	Version	Size	Single (ms)	Double (ms)
Titan Black	OpenCL	602	8.19	11.33	RadeonR9	OpenCL	602	4.89	10.10
Titan Black	LIFT	602	6.93	11.55	RadeonR9	LIFT	602	5.05	9.18
Titan Black	OpenCL	336	4.01	5.16	RadeonR9	OpenCL	336	2.93	4.91
Titan Black	LIFT	336	3.51	5.91	RadeonR9	LIFT	336	2.96	5.09
Titan Black	OpenCL	302	0.97	1.37	RadeonR9	OpenCL	302	0.60	1.19
Titan Black	LIFT	302	0.84	1.45	RadeonR9	LIFT	302	0.69	1.16
AMD7970	OpenCL	602	5.05	10.66	GTX780	OpenCL	602	9.21	12.30
AMD7970	LIFT	602	4.97	10.31	GTX780	LIFT	602	7.59	13.24
AMD7970	OpenCL	336	2.70	5.68	GTX780	OpenCL	336	4.57	5.65
AMD7970	LIFT	336	2.70	5.70	GTX780	LIFT	336	3.85	6.79
AMD7970	OpenCL	302	0.66	1.41	GTX780	OpenCL	302	1.23	1.52
AMD7970	LIFT	302	0.64	1.31	GTX780	LIFT	302	1.04	1.69

TABLE V: Median run time values for FI-MM data in Figure 5. Times are reported in milliseconds.

Platform	Version	Size	Shape	Single (ms)	Double (ms)
RadeonR9	OpenCL	602	box	0.28	0.51
RadeonR9	LIFT	602	box	0.28	0.35
RadeonR9	OpenCL	302	box	0.07	0.13
RadeonR9	LIFT	302	box	0.07	0.09
RadeonR9	OpenCL	336	box	0.32	0.60
RadeonR9	LIFT	336	box	0.33	0.37
AMD7970	OpenCL	602	box	0.27	0.34
AMD7970	LIFT	602	box	0.27	0.34
AMD7970	OpenCL	302	box	0.07	0.08
AMD7970	LIFT	302	box	0.07	0.08
AMD7970	OpenCL	336	box	0.29	0.33
AMD7970	LIFT	336	box	0.29	0.33
GTX780	OpenCL	602	box	0.27	0.33
GTX780	LIFT	602	box	0.27	0.34
GTX780	OpenCL	302	box	0.06	0.08
GTX780	LIFT	302	box	0.06	0.08
GTX780	OpenCL	336	box	0.25	0.34
GTX780	LIFT	336	box	0.25	0.34
Titan Black	OpenCL	602	box	0.29	0.31
Titan Black	LIFT	602	box	0.28	0.36
Titan Black	OpenCL	302	box	0.06	0.07
Titan Black	LIFT	302	box	0.06	0.09
Titan Black	OpenCL	336	box	0.30	0.29
Titan Black	LIFT	336	box	0.28	0.40
RadeonR9	OpenCL	602	dome	0.34	0.48
RadeonR9	LIFT	602	dome	0.34	0.37
RadeonR9	OpenCL	302	dome	0.08	0.11
RadeonR9	LIFT	302	dome	0.08	0.08
RadeonR9	OpenCL	336	dome	0.28	0.33
RadeonR9	LIFT	336	dome	0.28	0.27
AMD7970	OpenCL	602	dome	0.32	0.38
AMD7970	LIFT	602	dome	0.31	0.38
AMD7970	OpenCL	302	dome	0.08	0.09
AMD7970	LIFT	302	dome	0.08	0.09
AMD7970	OpenCL	336	dome	0.25	0.28
AMD7970	LIFT	336	dome	0.25	0.28
GTX780	OpenCL	602	dome	0.28	0.38
GTX780	LIFT	602	dome	0.29	0.38
GTX780	OpenCL	302	dome	0.06	0.09
GTX780	LIFT	302	dome	0.06	0.09
GTX780	OpenCL	336	dome	0.19	0.30
GTX780	LIFT	336	dome	0.21	0.30
Titan Black	OpenCL	602	dome	0.30	0.32
Titan Black	LIFT	602	dome	0.29	0.37
Titan Black	OpenCL	302	dome	0.06	0.07
Titan Black	LIFT	302	dome	0.06	0.08
Titan Black	OpenCL	336	dome	0.24	0.25
Titan Black	LIFT	336	dome	0.20	0.25

TABLE VI: Median run time values for FD-MM data in Figure 6. Times are reported in milliseconds.

Platform	Version	Size	Shape	Single (ms)	Double (ms)
RadeonR9	OpenCL	602	box	0.52	1.05
RadeonR9	LIFT	602	box	0.47	0.94
RadeonR9	OpenCL	302	box	0.12	0.26
RadeonR9	LIFT	302	box	0.12	0.23
RadeonR9	OpenCL	336	box	0.49	0.69
RadeonR9	LIFT	336	box	0.44	0.64
AMD7970	OpenCL	602	box	0.57	0.93
AMD7970	LIFT	602	box	0.54	0.85
AMD7970	OpenCL	302	box	0.13	0.22
AMD7970	LIFT	302	box	0.13	0.21
AMD7970	OpenCL	336	box	0.50	0.71
AMD7970	LIFT	336	box	0.47	0.69
GTX780	OpenCL	602	box	0.48	0.78
GTX780	LIFT	602	box	0.52	0.76
GTX780	OpenCL	302	box	0.11	0.18
GTX780	LIFT	302	box	0.12	0.18
GTX780	OpenCL	336	box	0.36	0.61
GTX780	LIFT	336	box	0.38	0.59
Titan Black	OpenCL	602	box	0.49	0.83
Titan Black	LIFT	602	box	0.50	0.87
Titan Black	OpenCL	302	box	0.11	0.20
Titan Black	LIFT	302	box	0.12	0.21
Titan Black	OpenCL	336	box	0.40	0.55
Titan Black	LIFT	336	box	0.40	0.60
RadeonR9	OpenCL	602	dome	0.45	0.66
RadeonR9	LIFT	602	dome	0.46	0.68
RadeonR9	OpenCL	302	dome	0.11	0.17
RadeonR9	LIFT	302	dome	0.11	0.17
RadeonR9	OpenCL	336	dome	0.37	0.41
RadeonR9	LIFT	336	dome	0.35	0.42
AMD7970	OpenCL	602	dome	0.48	0.70
AMD7970	LIFT	602	dome	0.48	0.70
AMD7970	OpenCL	302	dome	0.12	0.17
AMD7970	LIFT	302	dome	0.12	0.17
AMD7970	OpenCL	336	dome	0.36	0.47
AMD7970	LIFT	336	dome	0.36	0.47
GTX780	OpenCL	602	dome	0.41	0.60
GTX780	LIFT	602	dome	0.44	0.63
GTX780	OpenCL	302	dome	0.09	0.15
GTX780	LIFT	302	dome	0.10	0.16
GTX780	OpenCL	336	dome	0.29	0.45
GTX780	LIFT	336	dome	0.29	0.44
Titan Black	OpenCL	602	dome	0.42	0.56
Titan Black	LIFT	602	dome	0.43	0.65
Titan Black	OpenCL	302	dome	0.10	0.14
Titan Black	LIFT	302	dome	0.10	0.16
Titan Black	OpenCL	336	dome	0.30	0.36
Titan Black	LIFT	336	dome	0.30	0.42